The jointly scheduling of hard periodic tasks with soft aperiodic events within the Real-Time Specification for Java (RTSJ)

Damien Masson, Serge Midonnet

The jointly scheduling of hard periodic tasks with soft aperiodic events within the Real-Time Specification for Java (RTSJ)

Damien Masson[a,b], Serge Midonnet[a,c]

[a] *Université Paris-Est, LIGM – UMR CNRS 8049, France.*
[b] *ESIEE Engineering, Cité Descartes BP 99, 93162 Noisy-le-Grand Cedex, France.*
[c] *Université de Marne-la-vallée, 77454 Marne-la-Vallée Cedex 2, France.*

Abstract

The studied problem is the jointly scheduling of hard periodic tasks with soft aperiodic events, where the response times of soft tasks have to be as low as possible while the warranty to meet their deadlines has to be given to hard tasks. A lot of theoretical solutions have been proposed these past two decades but we are interested on the implementability of these solutions under the real-time specification for Java (RTSJ), without changing the scheduler. This led us to adapt the existing algorithms to operate at a user land level in the system, to propose some optimizations and counter measures in order to balance the lost of performances and finally to set up an approximate slack stealer algorithm specifically designed to take into account RTSJ restrictions. We propose new classes to extend the RTSJ API's to implement these mechanisms and some minor modification suggestions to existing ones as a feed back from our RTSJ experiences. We demonstrates the efficiency of the modified algorithms through extensive simulations and the implementability on available RTSJ compliant virtual machine by an overhead measure in real situation with the RTSJ JamaïcaVM from Aïcas. We also measure the overhead on LejosRT, an RTSJ compliant firmware for Lego Mindstorms NXT in development.

1. Introduction

Real-time system theory has historically focused on fully periodic task systems. This came from the need to have known activation model to ensure the respect of timing constraints. Modern applications however cannot be only composed of fully periodic tasks. Some part of the work, like user feed back, control, or switch to another operating mode are aperiodic by essence. These kind of work cannot has strict constraints, since its activation model is not known, but the system can try to serve it as fast as possible.

This leads the community to address the problem of jointly scheduling hard periodic tasks and soft aperiodic events. The objective is to conjointly ensure the deadline for periodic tasks, and minimize the response times of other tasks.

A first solution is to schedule all non-periodic tasks at a lower priority (assuming that the tasks are scheduled using a preemptive fixed priority policy). This policy is known as the *Background Servicing (BS)*. If it is very simple to implement, it does not offer satisfying response times for non-periodic tasks, especially if the periodic traffic is important.

To reduce these response times, the periodic task servers were introduced by Lehoczky et al. in [1]. A periodic task server is a periodic task, for which classical response time determination and admission control methods are applicable (with or without modifications). This particular task is in charge of servicing the non-periodic traffic with a limited capacity.

Several types of task server can be found in the literature. They differ by the way the capacity is managed. We can cite the *Polling Server* policy (*PS*), the *Deferrable Server* policy (*DS*), the *Priority Exchange* policy (PE) first described by Lehoczky et al. in [1] and developed in [2] and the *Sporadic Server* policy (SS) presented in [3].

In [4], Lehoczky and Ramos-Thuel propose the *Static Slack Stealer*, an algorithm to compute the slack: the maximal amount of time available at instant t to execute aperiodic tasks at the highest priority without endangering the periodic tasks.

This approach was first considered optimal in terms of minimizing the aperiodic tasks response times. It was proved in [5] that such an optimal schedule cannot be obtained by a non clairvoyant algorithm.

Unfortunately, the time and memory complexities of slack stealing approach are much too high for it to be usable. In his PhD thesis work [6], Davis proposes a dynamic algorithm to compute the exact available slack time. Then he demonstrates that the complexity is too high for a real implementation and proposes two approximate slack time computation algorithms, one based on a static approximation, *SASS*, and the other which is dynamic, *DASS*. The dynamic approach presents the advantage to allow gain time[1] to be assimilated in the slack and then transparently reallocated for aperiodic traffic.

All these contributions make the assumption that the developer has the hand on the scheduler, and are in substance enhanced scheduling mechanisms. Unfortunately, the scheduler is most of the time a part of the hardware or of the operating system.

In this work, we focus on the implementability of these mechanisms with the Real-Time Specification for Java (RTSJ). The aim of RTSJ is to design APIs and virtual machine specifications for the writing of real-time applications in Java language.

The only easily supported mechanism in the RTSJ is the *BS*. We then focus on the adaptation of existing algorithms to run at a user land level, since we want to propose a solution running on actual RTSJ compliant virtual machine: we can just rely on a preemptive fixed priority scheduler.

[1] if a periodic task has a worst case execution time greater than its mean execution time, most of its executions generate reserved but unused time called *gain time*

We also designed a new approximate slack stealer specifically conceived to take into account RTSJ implementation restrictions. Since the aim of this algorithm was to have the lowest possible overhead, we called it the Minimal Approximate Slack Stealer (*MASS*).

The remainder of this paper is organized as follow: Section 2 presents related work ; Section 3 presents the task model and the assumptions ; Section 4 describes the existing algorithms ; Section 5 presents the RTSJ existing tools, the constraints of their use, and the solutions to pass through these constraints and implement task servers ; Section 6 focus on the slack stealer implementations and the MASS algorithm ; Section 7 presents an unified framework to handle aperiodic traffic in the RTSJ ; Section 8 exposes simulation and execution results to evaluate our mechanisms and Section 9 regroups some suggestions to improve the RTSJ. Then we conclude in Section 10.

2. Context and related work

The context of this work is the jointly scheduling of hard periodic tasks with soft aperiodic events. The deadlines of periodic tasks must be ensured while the response times of aperiodic events may be minimized.

Our goal is not to propose new and better algorithms, but to study the implementability of existing ones.

Our target is the RTSJ since the research effort both from academy and industry is more and more consequent these past ten years. It looks for us as a good vehicle to speed up the technology transfer from academy to industry.

The choice of this target limited us to fixed priority systems, although results exists for the jointly scheduling problem in dynamic priority systems [7, 8]. The restrictions brought to us by the use of RTSJ also led us to propose a new slack time approximation algorithm (MASS). It's aim was not to propose a closer bound than existing algorithms (SASS, DASS), but to operate with a lower time overhead.

If this algorithm was designed in the context of the mixed scheduling, slack time approximation is an actual topic of interest in other domain, like the energy aware scheduling using modern processors able to dynamically adjust their frequency and voltage (DVS/DFS).

3. Task Model, Assumptions and Notations

We consider a process model of a mono processor system, Φ, made up of n periodic tasks, $\Pi = \{\tau_1, ..., \tau_n\}$ scheduled with fixed priorities. Each $\tau_i \in \Pi$ is a sequence of requests for execution characterized by the tuple $\tau_i = (r_i, C_i, T_i, D_i, Pi)$, where r_i is the instant of τ_i first release, C_i is the worst case execution time (WCET) of the request, T_i is the task period i.e. the amount of time between two consecutive requests, D_i is the deadline of a request relative to its release and P_i is the priority of the task, which can be arbitrary set (the priorities are not related to periods nor deadlines).

Since we focus on this paper on the cases where the execution time is constant over all requests, we will denote the WCET the *cost* of the request.

We restrict the study to the case $D_i \leq T_i$.

The highest priority is 1 and tasks are ordered according to their priority, so τ_i has a higher priority than τ_{i+1} and more generally we have: $P_1 < P_2 < ... < P_n$.

The system also has to serve an unbounded number p of aperiodic requests, $\Gamma = \{\sigma_1, ..., \sigma_p\}$. A request $\sigma_i \in \Gamma$ is characterized by a release date r_i and by an execution time C_i.

Finally we assume that all tasks are independent and so we do not consider any blocking factor nor release jitter. If this work can be extended to the case where periodic tasks can share resources and possibly to the case with jitter on the periodic activations, it cannot be extended to the case where aperiodic share resources (with other aperiodic tasks or with periodic tasks). We will see why in Section 5.2.3. This work can also be extended to the case where priorities are associated to aperiodic tasks. We will see how in Section 5.2.2.

We denote as an *i-level busy period* a time interval where the processor is always occupied by tasks with priorities higher or equal to i. Reciprocally, we denote as an *i-level idle period* a time interval where the processor is idle or occupied by tasks with priorities lower than i. Generally, the *running level* designates the priority of the task currently executed.

4. Existing Algorithms

The general problem of jointly scheduling hard periodic tasks and soft aperiodic events was widely studied through the past decades.

The first solution to this scheduling issue consists to serve the aperiodic traffic in background. This approach, called *background scheduling*, is easily set up by the reservation of a range of the lowest priorities to serve the aperiodic tasks, and this is sufficient to ensure that they will never interfere with the periodic traffic.

However, if this background scheduling offers the feasibility guarantee for the periodic tasks, it does not address the issue of the minimization of the aperiodic tasks response times.

We can classify the algorithms which address the whole issue in two main categories: the task servers and the slack stealers. In one hand the task servers approach is a resource reservation: a special task responsible to handle the aperiodic traffic is added to the feasibility analysis. On the other hand the slack stealer approach consists in the computation and the use as soon as possible of the unused resources of the system.

Task servers was first introduced in [1]. This paper presents the general concept and three possible algorithms: the *Polling Server (PS)*, the *Deferrable Server (DS)* and the *Priority Exchange Server (PES)*. All these three algorithms rely on a special task in the system with a given priority, a given period and a given capacity. This task is responsible for the aperiodic traffic service and its

interference on the other periodic tasks can be bounded independently of the aperiodic workload.

Slack stealer algorithms was introduced in [4] and extended to hard real-time sporadic tasks in [9]. The general idea is to compute at a time t where there is an aperiodic pending request, for each hard real-time task, a value, called the slack. This value corresponds to the amount of time the task can suspend its execution without missing its deadline. Then a time interval equal to the minimum among these values can be used to handle aperiodic traffic. The whole issue is then to compute this slack values. The first proposed approach was based on a table computed off line. This solution suffers to a big memory complexity issue due to the static table storage. Moreover, it was difficult to integrate in the slack notion several dynamic parameters, such as gain time or release jitter. So a dynamic exact approach was proposed in [10], and a dynamic approximate one to address the time complexity issue of the exact approach was proposed in [6]. This dynamic approximation relies on the computation of a lower bound on the slack values and is called *Dynamic Approximate Slack Stealer* (DASS).

It is interesting to note that the exact slack stealer approaches was first consider as optimal for the problem of the minimization of aperiodic response times (in the context of a preemptive fixed priority scheduler) before it was proved in [5] that such an optimal algorithm cannot be obtained without knowledge on the aperiodic task arrival model. Indeed, the authors showed that starting to serve an aperiodic request as earlier as possible does not necessary lead to minimized its response time.

4.1. Polling server (PS)

A polling server is an ordinary periodic task τ_s, with a priority P_s, a period T_s and a cost C_s. From a feasibility analysis point of view, it is absolutely identical to a normal periodic task. This task has access to a queue where the aperiodic events are enqueued when they are released. The policy gesture of this queue (*first in first out*, *last in first out*, any other order) is independent with the polling server general principle and does not interfere with the hard tasks feasibility analysis.

When the server is periodically released, its capacity is set to its cost parameter. Then it checks the emptiness of its job queue. If the queue is not empty, it begins to serve the jobs, consuming its capacity, until its capacity falls to zero. If it happens that there is no more job to serve, its capacity falls instantaneously to zero.

The principal draw back of this algorithm is that if it happens that an aperiodic job is released just after a periodic activation of the server, say at a time $t = \alpha T_s + \epsilon$ with α an arbitrary positive integer and if the queue was empty at that time, the service of this job will be delayed at least to the next activation ($(\alpha + 1)T_s$) since the server has just lost its capacity.

4.2. Deferrable Server (DS)

The DS was proposed to address this PS weakness. The DS algorithm is quite similar to the PS one, except that it keeps its capacity even if the queue

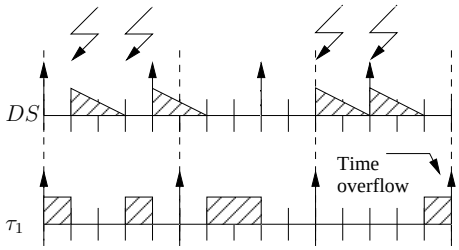


Figure 1: We have in this example a DS with $C_s = 2$, $T_s = 4$ and a periodic hard real-time task τ_1 with $C_1 = 2$ and $T_1 = 5$. If the system is analyzed as if the DS was a regular periodic task, it is found feasible. However, we can see that if an aperiodic request is released at time 10 when the DS has kept its capacity, and another one at time 12 when the DS just retrieved its full capacity, τ_1 cannot complete its periodic request before its relative deadline.

is empty. It results from this propriety that the DS can be woke up at any moment, and can preempt a lower priority task. If it permits to decrease the average response times of aperiodic events served, it also violates one hypothesis of the feasibility analysis theory which is that a periodic hard real-time task cannot suspends itself. Consequently, the interference of the DS on the other tasks of the system is not the same as the one from a periodic task with the same parameters. To illustrate that, an example is presented in Figure 1.

The direct consequence is that number of systems feasible with a PS will not be feasible with a DS set with the same parameters. The cost to pay to obtain lower average response times for aperiodic tasks is a lower feasibility bound on the periodic ones.

4.3. Dynamic Approximate Slack Stealer (DASS)

We describe here with details the Davis work on the dynamic slack stealing in order to understand well the common parts and the differences with our algorithm that we describe in Section 6.2.

4.3.1. Dynamic Slack Stealing (DSS)

One goal of this dynamic approach was to extend the slack stealing algorithm to systems with hard sporadic tasks. In such systems, it is a strong limitation to only consider the execution of aperiodic requests at the higher priority. So the original DSS allows to compute the higher priority to execute the aperiodic request and the amount of time associated. For the sake of clarity and due to an RTSJ limitation we will present in Section 5.2, we just present here the algorithm version where aperiodic requests are scheduled at the highest priority.

We consider an aperiodic request, J_a, released at time t. DSS is an algorithm to schedule this request. This algorithm relies on the determination at time t and for each priority level of the available slack time, $S_i(t)$, which is the maximum amount of time the task τ_i can be delayed without missing its deadline. This

value is equal to the number of unused time units at priorities higher or equal to i between t and the next τ_i deadline. the length of this interval is noted $d_i(t)$.

Then, in order to serve J_i at the highest priority while guaranteeing the periodic tasks deadlines, we need to have a positive value for all $S_i(t)$. The number of "stolable" time units i.e. immediately available is then $S(t) = \min_{\forall i} S_i(t)$.

To compute the $S_i(t)$ values, the interval between t and the next τ_i deadline that we denote $[t, t + d_i(t))$ is viewed as a succession of *i-level* busy periods[2] and *i-level* idle periods[3]. Then, $S_i(t)$ is the sum of the *i-level* idle period lengths. Equations to compute the end of a busy period starting at time t and the length of an idle period starting at time t can be derived from the feasibility analysis theory. These two equations are then recursively applied until the reach of the next deadline to determine $S_i(t)$.

With this methodology, slack time has potentially to be recomputed from scratch at any instant, which is obviously not practicable. In order to reduce the complexity of the computation, we have to define slack time at time t' relatively to slack time at time t. There is two general cases to study:

None of the hard periodic tasks ends its execution in $[t, t')$. Then there is three possibilities for the processor activity between t and t': the processor can be idle, it can be executing soft aperiodic requests (stealing slack times) and it can be executing hard periodic tasks. For the two first possibilities, the slack is reduced by $(t' - t)$ for all priorities. However, if the processor is executing the hard real-time task τ_i, then the system is idle for higher priorities $k < i$, and the slack is reduced by $(t - t')$ only for these priorities.

One periodic hard real-time task, τ_i ends its execution at time $t'' \in [t, t')$. Then all *i-level* idle times present in $[t, d_i(t))$ will be also present in $[t, d_i(t) + T_i) = [t, d_i(t''))$, which is the new interval to consider for the *i-level* slack times computation. Therefore, the τ_i termination can only increase $S_i(t)$ but never decrease it. Consequently, $S_i(t)$ has to be recomputed each time τ_i ends a periodic activation.

So, assuming that there is a time t where the $S_i(t)$ was up to date for all tasks, the DSS algorithm to compute $S(t'')$ is :

1. if none of the periodic hard real-time tasks ends in $[t, t')$
   (a) if the processor is idle or executing soft aperiodic requests
   $$\forall j : S_j(t') = S_j(t) - (t - t') \quad (1)$$
   (b) if the processor is executing hard periodic task τ_i
   $$\forall j < i : S_j(t') = S_j(t) - (t - t') \quad (2)$$
2. if hard real-time task τ_i ends at time $t'' \in [t, t')$, $S_i(t'')$ has to be computed using the recursive analysis described at the beginning of this section. The recurrence can be started with the previous value ($S_i(t'') = S_i(t)$).

[2] periods where the processor is servicing priorities higher or equal to i

[3] processor idle periods or periods where processor serves priorities lower than i

This algorithm is not directly usable because of the time complexity of the recursive computation of the $S_i(t)$ to perform at each task ends. However, this part can be replaced by the computation of a lower bound.

4.3.2. Dynamic Approximate Slack Stealer (DASS)

Since $S_i(t)$ is the sum of the i-level idle period lengths in the interval $[t, t + d_i(t))$, DAVIS proposes to estimate this quantity by computing a bound on the maximal interference the task τ_i can suffer in this interval. A bound on this interference is given by the sum of the interferences from each task with a higher priority than τ_i. Then Equation 3 gives the interference suffer by a task τ_j from a task τ_i in an interval $[a, b]$.

$$I_i^j(a,b) = c_i(t) + f_i(a,b)C_i + \min(C_i, (b - x_i(a) - f_i(a,b)T_i)_0) \tag{3}$$

The function $f_i(a, b)$ returns the τ_i instance number which can begins and completes in $[a, b]$. It is given by Equation 4[4].

$$f_i(a,b) = \left\lfloor \frac{b - x_i(a)}{T_i} \right\rfloor_0 \tag{4}$$

The function $x_i(t)$ represents the first activation of τ_i which follows t. Then the interference is composed by the remaining computation time needed to complete the current pending request, by a number of entire invocations given by $f_i(a, b)$, and by a last partial request.

A lower bound on the $S_i(t)$ value is given by the length of the interval minus the sum of the interferences from each task with a higher or equal priority than τ_i. It is recapitulated by Equation 5.

$$S_i(t) = \left(d_i(t) - t - \sum_{\forall j \leq i} I_j^i(t, d_i(t)) \right)_0 \tag{5}$$

4.4. Conclusions

The jointly scheduling of hard periodic tasks with soft aperiodic events issues was widely studied from a theoretical point of view.

The task server approache consists to delegate the aperiodic traffic to a special task and to modify the feasibility analysis algorithm to integrate this special task. The DS offers better responsiveness for aperiodic traffic, but results in a lower feasibility bound due to its greater interference on task with a lower priority. Other service politics exists, we can cite here the sporadic server or the priority exchange server, but are more complicated to set up.

[4] the notation $(x)_0$ means $max(x, 0)$

The slack stealer approache does not modify the analysis of the periodic tasks, but tends to stole unused time which results from the necessity to guarantee the periodic tasks deadlines in the worst case. It offers the best performances among the non forth-seeing algorithms since it allows to start an aperiodic request as soon as possible at the best priority.

The common part between these two approaches is that they are strongly linked with the scheduler, and even are in essence scheduling algorithms.

5. RTSJ Existing Tools, Constraints and Algorithmic Solutions

If the JSR-01 for a real-time specification for Java is completed, RTSJ is still a young specification, with a lot of features missing and in progress through the JSR-282. So on, the expert group focus his attention to issues such as cross-platforms time facilities, real-time threads or non garbage-collected memory areas.

Some tools are provided to model aperiodic traffic, but we will see in this section that they do not permits to set up advanced mechanisms presented in the last section.

5.1. Existing tools in the RTSJ

The RTSJ proposes a set of minimal requirements for a real-time Java virtual machine (RTJVM) and a set of API to program real-time applications in Java. Some features are mandatory, some others are not. Since our will is to propose a portable mechanism, we only focused on mandatory features.

In real-time systems, the scheduling policy has to be a real-time policy. So the RTSJ specifies that an RTJVM has to offer by default a preemptive fixed priority scheduler. It allows the programmer to choose an alternative policy, by the way of the abstract class `Scheduler`. However, a very few RTJVMs implements other policies, and it is difficult to write a scheduler independent of the JVM.

To generalize the `Thread` facility of regular Java programs, RTSJ proposes the notion of schedulable object (SO) through the interface `Schedulable`. A SO is an object which can be scheduled by the scheduler. A suitable context for the execution of such an object is created using scheduling and release parameters attached to each SO. `Schedulable` is so an interface with getter and setter methods to access and modify scheduling parameters (class `SchedulingParameters`) and release ones (class `ReleaseParameters`). There is also methods to attach the SO to a particular scheduler or to add or remove it to the feasibility analysis.

Two main concrete classes implementing this interface are proposed: `RealtimeThread` which also extends the class `java.lang.Thread`, and `AsyncEventHandler` which is a lighter way to encapsulate code in order to execute it when some event is released. A `RealtimeThread` is always associated with one and only one thread, but `AsyncEventHandler`s can share a thread if a thread pool mechanism is implanted (this is a possibility offered by the specification but this is not mandatory).

So `RealtimeThread`s associated with `PeriodicParameters` (which is a `ReleaseParameters` subclass) are perfect to implement hard real-time periodic tasks, and `AsyncEventHandler`s seems to be the right choice to implement soft aperiodic ones.

However, among the solutions for a mixed scheduling of these two kind of traffic, only the BS is implementable with this tools. It is sufficient to assign `PriorityParameters` (`SchedulingParameters` subclass) with lower priorities to the `AsyncEventHandler`s which model the aperiodic traffic. It is possible to go further by the use of `ProcessingGroupParameters` (PGP). A PGP is an object quite similar to `ReleaseParameters`, but shared by several SO. It allows in particular to assign a common time budget to several SO. So it permits to virtually set up a server at a logical level. Unfortunately, the desirable behavior when the budget is consumed, which is to not schedule the others SOs until the budget is replenished, is not mandatory because it relies on the capacity to monitor CPU consumption of the JVM. Moreover, this behavior is underspecified [11] and in any case does not allow to choose a particular policy for the consumption of the shared budget. A contribution to extend the PGP semantic for multiprocessors systems and to extend it in order to write task servers has been led in [12]. However this approach relies on modifications on the specification and is not implantable under nowadays available RTJVMs.

5.2. Goal and Restrictions

Our goal is to propose a framework to handle aperiodic traffic implementable without modification on the RTJVM or the RTOS. It supposes to only rely on mandatory features and to not modify the scheduler. The solutions we want to propose also have to present a reasonable overhead, which can be integrated in the feasibility analysis process.

Since we cannot modify the system scheduler and since the mechanisms are by essence scheduler, we will have to simulate a scheduler within a user land task.

Unfortunately, Java does not permits to asynchronously pause a thread from another thread. The RTSJ brings the asynchronous transfer of control (ATC) facility through the classes `AsynchrounouslyInterruptedException` and `Interruptible`, but it only permits to stop a thread definitively.

The second main issue is that both server mechanisms and slack stealer approach needs to monitor CPU time consumption. Indeed, even simplest servers like polling or deferrable needs to monitor the capacity consumption, and both slack stealer approach and more sophisticated server policies like Priority Exchange [1] or Sporadic Server [3, 13] suppose to keep up to date information about times passed to execute each priority level in order to compute data used by the algorithm.

The only solution we found to this issue is to force the server priority, or the priority level for aperiodic tasks executed on stolen slack time to the highest priority available. Then, if we use for the other tasks (regular periodic hard real time ones) a modified `RealtimeThread` class which perform special operations

when they begin and end each periodic job, we can easily trace the CPU occupation as we will explain in details in Section 5.2.1. We will see in Section 6 that the possibility to add code before and after each instance is also very convenient to implement slack stealer algorithms.

It is a strong and inelegant restriction since it does not permit to use several servers at the same time and it forces the use of modified `RealtimeThread`. Moreover, if one add in the system a regular `RealtimeThread`, there is no mechanisms to warn him he is doing something wrong. However, a better way to implement task servers and slack stealers only passed by modifications on the RTSJ.

5.2.1. Server Capacity Gesture

The cost enforcement facilities of the RTSJ are not mandatory. So the only solution to our knowledge to watch the CPU consumption of a task is to ensure that it is not preempted, and to measure time before and after its execution. The simplest way to forbid the preemption is to schedule it at the highest priority. We also could use a lock shared between all the tasks in the system, but we do not choose this solution because of the overhead of lock management.

Also, that suppose there is no task in the system with higher priority than the RTJVM.

So the server is a real-time task with the highest priority in the system. It has access to a structure where events are enqueued when they are released. When it is waked up, it starts a timer with its capacity, and starts to serve one aperiodic job. If the timer expire when it is still occupied with an event, it has to asynchronously stop the event. If it happens what the queue is empty, the timer is canceled and the server wait until its next activation.

There is two consequences: first the two highest priorities have to be reserved for the mechanism. The highest for the timers, and the second highest for the service. Second, it can happens what we start to treat an event for nothing, because the treatment is stopped asynchronously by the timer, and then the treatment will have to be resume from scratch at the next server activation. Even more, it is possible that an event never can be served, whereas it could be served in background or in a system where it is not scheduled at user land level.

Two actions can be set up to limit the side effect: take a special care to the queue policy, and schedule a replica in background. This is the topic of the next two subsections.

5.2.2. Waiting Queue Policy

To avoid starvation situations, the first action that we can take is to privilege shorter tasks, and to not start a task if there is not enough capacity to complete it. We investigated among several queue policy and conducted extensive simulations that shown that the *Lowest Cost First* (LCF) policy is almost always the policy which exhibits the shorter average response times.

However, if it decreases the risk of starvation, it does not solve the issue. It is still possible to never be able to serve a task, even if there is actually a lot of unused CPU time slot in the system, as shown by Figure 2.

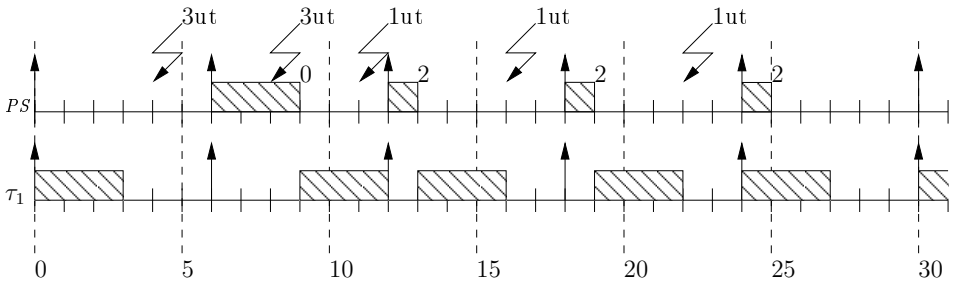


Figure 2: In this example, the event released at time $t = 8$ can never be served even if the server capacity never fall in a lower value than 2 after time $t = 12$

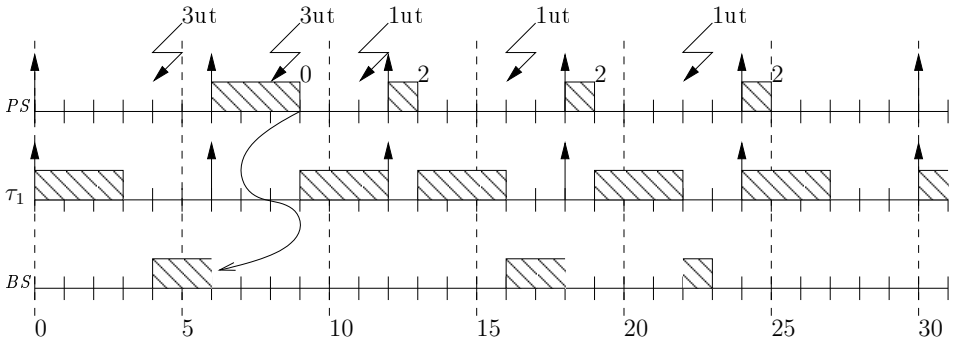


Figure 3: The same example as in Figure 2 but with a duplication policy set: the event which was never served without duplication is served in background and completes at time $t = 23$; the first aperiodic event that occurs at time $t = 4$ is first started in background but finally completes within the serer: the replica is asynchronously interrupted

5.2.3. Duplication in Background

The solution is to duplicate each aperiodic task and to schedule one replica in background. The first replica to complete will be in charge to cancel the second one. It is important to note that the replica will never compete against each over because the replica served by the manager will always be executed at the highest priority. However, it can happen that the background replica begins, then is preempted by other task, and then the other replica is started and complete its execution. Some job is then done twice, but if we do not consider energy consumption issue, it is not a problem since this CPU time would be wasted anyway. This is illustrated by Figure 3. Note that duplication can only be applied if the handler does not try to enter in a critical section.

5.3. Task servers in user land

We call respectively MPS and MDS the PS and DS algorithms modified in order to be implemented in user land within the RTSJ. The differences are:

- MPS and MDS must run at the highest priority,

- they start a task only if there is still enough capacity in the server to permits its termination.

6. Slack Stealers in user land

We explain in this section how the DASS algorithm can be adapted in order to be implemented as a server running in user land. We introduce a new algorithm, MASS, which is specifically conceived to run in such environment. The MASS algorithm reduces the time complexity of the slack bound computation, but the cost is paid with the quality of this bound.

6.1. Modified-DASS (MDASS)

The DASS algorithm formulates $S_i(t)$, the lower bound on the slack time which can be stolen at priority i at time instant t, as the duration of the interval $[t, d_i(t))$ minus the maximal interference caused by higher priority level tasks. This interference can be bounded by the sum of the interference caused by each higher priority level task took individually.

Then, Equation 3 gives us the interference of a given priority level during a given interval. Summing the interferences from all higher priority level gives us a bound on the total interference and as a consequence a bound on $S_i(t)$. Then we just have to take the minimum to obtain $S(t)$.

It has to be noted that for a given task τ_i, $S_i(t)$ only has to be computed when task τ_i ends a periodic instance. Rest of the time, it does not change if the running level is higher or equal to i, and it decreases when it is lower.

So, it seems that CPU time monitoring is mandatory in order to implement DASS: first because $c_i(t)$ is used in Equation 3, second in order to decrease $S_i(t)$ when the running level is lower than i.

CPU-Time for interference approximation. Equation 3 gives us the interference of a task τ_i on another task τ_j in an $[a, b]$ time interval, with $i < j$.

$$I_i^j(a,b) = \bar{\mathbf{c}}_\mathbf{i}(\mathbf{a}) + f_i(a,b)C_i + \min(C_i, (b - x_i(a) - f_i(a,b)T_i)_0)$$

It seems that the remaining execution time $cb_i(t)$ has to be known. In fact, is this computation is only done when τ_j ends an instance, it implies that $\bar{c}_i(t)$ equals 0. Indeed, in the hypothesis that this value is not null, then τ_i should be the executing task, not τ_j.

CPU-time to update $\mathbf{S_i(t)}$. When the value of $S_i(t)$ has to be know, it is sufficient to subtract to the last computed value the time consumed in the elapsed time by all the lower priority tasks, including the processors idles time.

If the operating system does not provide these date, it is still possible to manually watch the consumption of each task. In order to do that, a stack has to be updated. When a task starts, it is pushed on the stack, and when it ends, it is popped. Before pushing a task on the stack, we update the consumed time of the task on the top of the stack.

It is then possible, each time that a task starts, to update the value of $S_i(t)$ for all the task with an higher priority than the one previously executing. On same way, at any instant t, it is possible to update the value for the tasks with an higher priority than the one which is executing.

Conclusion. It is so possible to implement DASS in user land, to the price of adding at before each task start an uptade in linear time of all the $S_i(t)$, and at the end of each time, a linear time computation of the $S_i(t)$ for the ending task. These operations has to be protect against interruption. We the actual slack time in the system is needed, it is necessary to update all the $S_i(t)$ values and to compute the minimum. It is also a linear time complexity operation.

However, a roughly slack time approximation can be obtained in constant time. Since all the task are examined at the end of each instance, it is possible to extract the minimum over the $S_i(t)$ in the same process. After that, the worst scenario can be assumed, which is the one where the slack is continuously consumed between the last minimum computation and the instant t. This can be optimized if the priority level of the task with the lower $S_i(t)$ is kept. Then, we can compare it to the priority of the executing task when the slack is needed: if the executing task has a lower priority then the slack has decreased else the slack has kept its value.

6.2. MASS

We propose a new algorithm, MASS, with a lower overhead on the system than DASS. The goal is to reduce the impact on the schedulability of the system. MASS stands for Minimal Approximate Slack Stealer. As DASS, MASS relies only on operation added at the beginning and at the end of each periodic job.

The time complexity of the operation added at the end is still linear, but with a lower constant. However the time complexity of added operation at the beginning is now constant, and these operations are only needed if the system does not provided the CPU consumption of each task.

Notations. The notations used to describe the DASS algorithm are reused here.

We note α_i the aperiodic task number i. The highest priority level is 1. A 0 level activity corresponds to the system being idle.

6.2.1. Data to collect and compute

We decompose $S_i(t)$ in two parts to be able to compute its separately. First we consider the available work at a given priority before the next deadline. This quantity is denoted $\bar{w}_i(t)$. Second we consider the work demand at a given priority, denoted $\bar{c}_i(t)$.

We then have $S_i(t) = \bar{w}_i(t) - \bar{c}_i(t)$.

Equation 6 then gives us the slack time in the system.

$$S(t) = \min_{\forall i \in hrtp} S_i(t) = \min_{\forall i \in hrtp} (\bar{w}_i(t) - \bar{c}_i(t)) \tag{6}$$

We can note that since the slack times are only computed when a periodic job completes its instance, the $\bar{c}_i(t)$ values only have to be correct at these instants.

6.2.2. Data initialization

In the hypothesis of a synchronous activation of periodic tasks, the equation 7 permits to compute for each task an upper bound on the interference suffer by τ_i from tasks with higher priorities.

$$I_i(t_0) \leq \sum_{\forall k<i} \left\lceil \frac{D_i}{T_k} \right\rceil C_k \tag{7}$$

This upper bound is given by the sum of interferences caused by each task with an higher priority taken separately. We obtain a value more pessimistic than the one obtain with Equation 3, but this will permits us to estimate more easily the interference for the next instance. The interference counted here too early will permits us to consider only the interference a task suffer between its activation and its deadline.

If the first activation is not synchronous, we just have to add to the interference suffer by a task τ_k the cost of each task with an higher priority started before it.

Consequently, we have when the system starts:

$$\forall i \begin{cases} \bar{w}_i(t_0) &= D_i - I_i(t_0) \\ \bar{c}_i(t_0) &= C_i \end{cases} \tag{8}$$

The time complexity is quadratic but this is not an issue since we can assume that this is done before the start of the system.

6.2.3. Data update

We note δ_k the duration of the processor occupation by τ_k within a given interval $[t_1, t_2]$. During the interval,$\bar{c}_k$ has decreased by δ_k, and for all task τ_k with an higher priority than τ_k, $\bar{w}_l$ has decreased by δ_k.

However, accurate values for $\bar{w}_i$ and $\bar{c}_i$ are only needed when a task completes one instance. We show here how it is possible to maintain a lower bound on $\bar{w}_i$ for all task with linear time operations added at the end of each periodic instances, and a higher bound on $\bar{c}_i$ for all task with constant time operations added at the end and at the begin of each periodic instances.

Special notations for the rest of this section:. We note t_b the latest time where a task has begin an instance and t_e the latest time where a task has end an instance. The current time is t and we have $dt_e = (t - t_e)$ and $dt_b = (t - t_b)$.

6.2.4. Lower bound on all $\bar{w}_i$

We consider the hard real-time periodic task τ_l. τ_l ens an instance at time t. The last update for all $\bar{w}_i$ values was at time t_e. We so have to study the interval $[t_e, t]$. We want to obtain for all task τ_k the value $\bar{w}_k(t)$ as a function of $\bar{w}_k(t_e)$.

Tasks with a priority higher than the one of τ_l. For all tasks τ_k with a priority higher than τ_l, we have $\bar{w}_k(t_e) = \bar{w}_k(t_b) - dt_e$ since only task with lower priority can have execute during interval $[t_e, t]$. Indeed, if a task with an higher priority had been executed in the interval, it should have ended its execution before τ_l, since it has an higher priority.

$$\forall k \in hrtp \backslash k < l, \bar{w}_k(t) = \bar{w}_k(t_e) - dt_e$$

τ_l. τ_l has just ended one instance. Since the last periodic task end, the system only can have schedule τ_l or some other tasks with lower priority. Consequently, $\bar{w}_l$ has decreased by dt_e. However, the considered interval for the l-level slack time computation is increased by T_l. The $\bar{w}_l$ value is so increased by T_l minus the interference the task will suffer from tasks with higher priority activated between the next τ_l activation, $x_l(t_e)$, and its next deadline $d_l(t_e)$. The interference suffer before $x_l(t_e)$ has already been included in the current value of $\bar{w}_l(t_e)$. A procedure to obtain in constant time the interference suffered by a task between one of its activation and its relative deadline is given in Section 6.2.7.

$$\bar{w}_k(t) = \bar{w}_k(t_e) - dt_e + T_k - I_k(t)$$

Taks with lower priority than τ_l. For tasks with a lower priority than τ_l, the issue is more complicated. Indeed, the available work as decreased only for tasks with higher priority than the ones effectively executed within the interval. The solution in DASS is to update (with a linear time complexity) the $\bar{w}_i$ at each instance begin. We propose here to tolerate a temporary lost of precision. We consider that the available work has decreased by dt_e for all task with a priority lower than τ_l. Nevertheless, we also immediately add to $\bar{w}_i$ the time consumed within the interval, $\bar{c}_l(t)$, since this value was considered in the interference suffer by each τ_i but now the task has ended and cannot interfere anymore. The remainder of the error will be corrected when the tasks really executed in the interval will end there current instance (by in the increase of the available work by their effective execution time for all task with a lower priority).

$$\forall k > l, \bar{w}_k(t) = \bar{w}_k(t_e) - dt_e + \bar{c}_l^*$$

where $\bar{c}_l^*$ is the time consumed by τ_l. It is equal to $C_l - (\bar{c}_l(t) - \min(dt_e, dt_b))$ if the model permits variable execution times, C_l if this is not the case.

6.2.5. Update of an $\bar{c}_i$ upper bound

To keep up to date the $\bar{c}_i(t)$ values, it is necessary to perform computations at each instance begin and end. The procedure is quite similar of the DASS one, but we only update $\bar{c}_l(t)$ where τ_l is the task occupying the processor, which is done in constant time. The $\bar{w}_i(t)$ update is delayed until the next task end.

So when a task τ_l begins an instance at time t, if k is the priority level of the executing task at time $t - \epsilon$, $\bar{c}_k$ is decreased by $\min(dt_e, dt_b)$.

When a task τ_l ends a periodic instance at time t, $\bar{c}_l(t) = C_l$.

6.2.6. Conclusion: operations to perform

When the periodic real-time task τ_l ends an instance:

$$\begin{array}{rl} \forall k < l, & \bar{w}_k(t) = \bar{w}_k(t_e) - dt_e \\ \forall k > l, & \bar{w}_k(t) = \bar{w}_k(t_e) - dt_e + \bar{c}_k^* \\ k = l, & \left\{ \begin{array}{rcl} \bar{w}_k(t) & = & \bar{w}_k(t_e) - dt_e + T_k - I_k(t) \\ \bar{c}_k(t) & = & C_k \end{array} \right. \end{array} \tag{9}$$

These operations can be performed with a linear time complexity.

When the periodic real-time task τ_l begins an instance:

Let τ_k be the task previously executed by the processor (if it was occupied by a periodic real-time task),

$$\text{If } k \in hrtp, \bar{c}_k(t) = \bar{c}_k(\max(t_e, t_b)) - \min(dt_e, dt_b) \tag{10}$$

This operation can be performed with a constant time complexity.

6.2.7. Interference approximation

We consider a periodic real-time task τ_j ending on periodic instance at time t_2. We know $\bar{w}_j(t_1)$, t_1 being the time of the last $\bar{w}_j$ update. We note its next deadlin $d_j(t_2)$ and the next one $d_j^+(t_2)$. We have $d_j(t_2) \geq t_2$ and $d_j^+(t_2) = d_j(t_2) + T_j$. We know that between t_1 and t_2 the work available at priority j has decreased by $dt = t_2 - t_1$. At time t_2, it increases by T_j minus $I_j(t_2)$, the interference suffers by τ_j from tasks with higher priority activated between $d_j(t_2)$ and $d_j^+(t_2)$. Indeed, the interference suffers from tasks activated before $d_j(t_2)$ has already been taken into account during $\bar{w}_j(t_1)$ computation.

The two following equation permits us to find an upper bound on this interference:

$$I_j(t) \leq \sum_{k<j} I_k^j(t) \tag{11}$$

$$I_i^j(t) \leq Nba_i^j(t).C_i \tag{12}$$

where $I_i^j(t)$ is the interference from τ_i on τ_j and $Nba_i^j(t)$ the number of tau_i activation between $d_j(t)$ and $d_j^+(t)$.

The first one derive from the fact that the interference suffered from the set of tasks with an higher priority is bound by the sum of each interference taken separately. The second says us that the interference from a task is bound by its number of activations multiply by its worst case execution cost.

We will show that $Nba_i^j(t)$, and so $I_i^j(t)$, can only take two different values if we study an interval of fixed length (T_j in MASS algorithm). Moreover, we can compute this two values off-line and choose the correct one on-line with a constant time complexity operation.

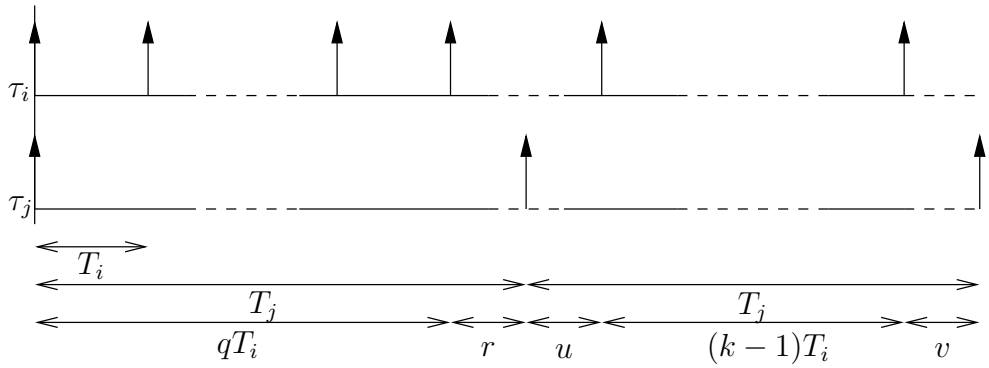


Figure 4: Number of activations – notations

6.2.8. Possible values for $\mathbf{Nba_i^j(t)}$

Let q and r be the quotient and the remainder in the euclidean division of T_j by T_i. We have $T_j = qT_i + r$. For each tau_j activation, we note u the time interval before the next τ_i activation, $k = Nba_i^j(t)$ the number of τ_i activations before the next τ_j activation and v the number such that $v = T_j - (k-1)T_i - u$. We have $T_j = u + (k-1)T_i + v$, $u < T_i$ and $v < T_i$.

Figure 4 summarizes this notations.

Theoreme 1. *There is only two different possible values for k which are $q+1$ and q.*

$k > q - 1$.
suppose $k \leq q - 1$,
$k \leq q - 1 \Rightarrow k - 1 \leq q - 2 \Rightarrow$
$u + (k-1)T_i + v < T_i + (q-2)T_i + T_i$
since $u < T_i$ and $v < T_i$.
So, since we have $u + (k-1)T_i + v = T_j$, we have $T_j < qT_i$.
which is in contradiction with $T_j = qT_i + r$, $r \geq 0$.

□

$k < q + 2$.
suppose $k \geq q + 2$,
$k \geq q + 2 \Rightarrow k - 1 \geq q + 1 \Rightarrow$
$u + (k-1)T_i + v \geq u + (q+1)T_i + v \Rightarrow$
$T_j \geq (q+1)T_i$, since $u \geq 0$ et $v \geq 0$. which is in contradiction with $T_j = qT_i + r$, $q \geq 1$ and $r < T_i$. □

6.2.9. Choosing the correct value for $\mathbf{Nba_i^j(t)}$

The number of τ_i activations within the interval $[d_j(t), d_j^+(t)]$ is equal to q or $q+1$. We will show that a simple comparison between u and r is sufficient to decide.

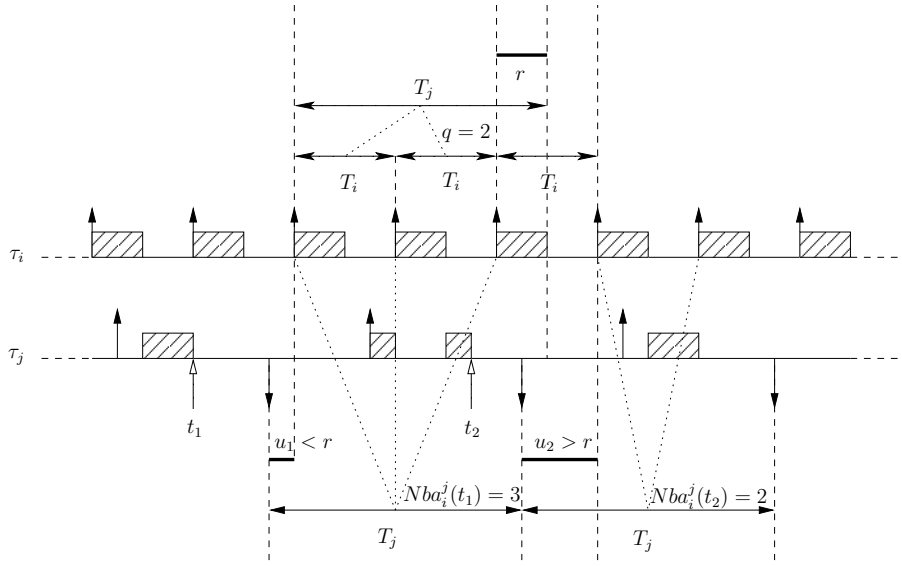


Figure 5: Number of activations – example

Theorem 2.

$$u \leq r \quad \Rightarrow \quad k = q + 1 \tag{13}$$

$$u > r \quad \Rightarrow \quad k = q \tag{14}$$

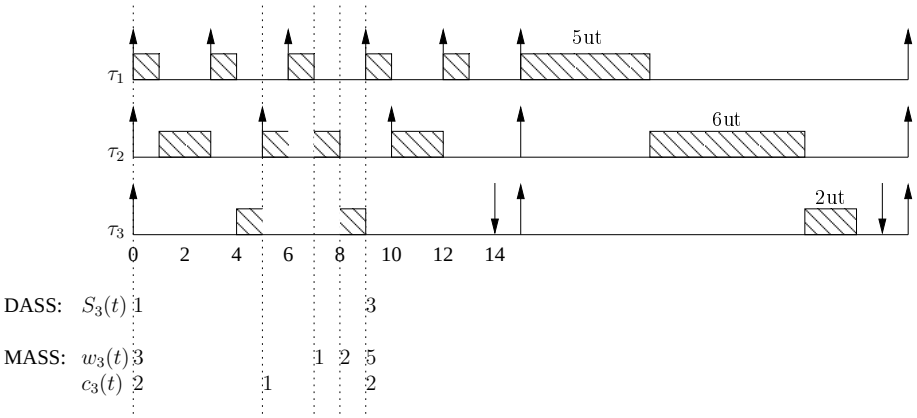
Equation 13.
$k = q + 1 \Rightarrow k - 1 = q \Rightarrow$
$u + (k-1)T_i + v = qT_i + u + v \Rightarrow$
$u + v = r \Rightarrow$
$u \leq r$ since $u \geq 0$ and $v \geq 0$. □

Equation 14.
$k = q \Rightarrow$
$u + (k-1)T_i + v = (q-1)T_i + u + v \Rightarrow$
$u + v - T_i = r \Rightarrow$
$u > r$ since $v < T_i$. □

Figure 5 illustrates this theorem. To determine $I_i^j(t)$, it is sufficient to compare u and r. Note that r can be computed before the start of the periodic tasks. To obtain u, we have to subtract $x_j(t)$ to $x_i(x_j(t))$: $x_j(t)$ is known and $x_i(x_j(t))$ can be obtained by a division and a multiplication ($x_i(t) = \lceil t/T_i \rceil T_i$). If the time complexity is the same that the one to obtain the exact interference (Equation 3), the number of operations is much less important.

	Period	Cost	Deadline
τ_1	3	1	3
τ_2	5	2	5
τ_3	15	2	14

(a) System data



(b) Task execution whithout aperiodic job

Figure 6: Slack time computation at priority level 3 with DASS and MASS

6.3. DASS and MASS Comparative Example

We propose here a numerical example to better understand the differences between DASS and MASS.

The studied system is composed of three tasks. The task parameters and the normal execution of the system when there is no aperiodic event to deal with are given by Figure 6.

We focus on the computation of slack time for the priority level 3.

With DASS. At time instant 0, $S_3(0)$ is equal to τ_3 deadline from which we subtract its cost and the interference of τ_1 and τ_2 in the time interval $[0, 14]$. We have $S_3(0) = 14-5*1-3*2-2 = 1$. It is th number of unused times slot at priority level 3 in this interval when the system does not accept any aperiodic job. The value does not evolute until time instant 9 since only task with a lesser or equal priority are executed. It has to be noted that at time instants 3 ans 5, it is necessary to subtract 2 tu to $S_1(t)$ and 1 tu to $S_1(t)$ and $S_2(t)$. What's why the time complexity of start-task computation are in linear time complexity.

At time instant 9, τ_3 ends the execution of its first instance. The time interval considered for the computation of $S_3(t)$ is now $[t, 29]$ and not $[t, 14]$. The duration of this interval at time instant $t = 9$ is 20 tu. We subtract from this value the interference of 4 activations of τ_2, of 7 activation of τ_1 and of 1 activation of τ_3. We have so $S_3(9) = 20 - 8 - 7 - 2 = 3$. $S_3(9)$ is recomputed

from scratch without used an information already available : there is 1 unused time unit before time instant 14.

With MASS. The $\bar{w}_3(0)$ value is equal to the τ_3 deadline from which we subtract the interferences of τ_1 and τ_2 on the interval $[0, 14]$. We so have $\bar{w}_3(0) = 14-6-5 = 3$. Since $\bar{c}_3(0) = 2$, it gives us $S_3(0) = 1$, ie the same value obtained with DASS.

At time instants 1 and 3, tasks τ_1 and τ_3 end an instance. $\bar{w}_3(t)$ is then decreased by the elapsed time, but immediately incremented by the cost of ending tasks, and so is constant until time instant 7.

This is not the case for $\bar{c}_3(t)$, which is updated at time instant 5. Indeed, τ_2 starts then an instance, and the previously executing task, τ_3 is updated. We have $\bar{w}_3(5) = 3$ and $\bar{c}_3(5) = 1$, which gives us $S_3(5) = 2$. This value is incorrect and is greater the correct one ($S_3(5) = 1$). However this error has no effect since slack times are not evaluated when a task ends its execution. The next evaluation will occure at time instant 7.

At this time, $\bar{w}_3$ and $\bar{w}_2$ are decreased by 3 tu, because its corresponds to the elapsed time since the last end of a periodic task, and increased by the cost of τ_1, ie only 1 tu. We so have $S_3(7) = 3 - 3 + 1 - 1 = 0$, which a lesser value than the one obtained with DASS.

At time 8, τ_2 also ends a periodic execution. $\bar{w}_3(8)$ is so decreased by 1 tu and increased by 2 tu. We so have $S_3(8) = 1 + 2 - 1 - 1 = 1$, which is the same value than the one obtained with DASS.

Finally, at time 9, τ_3 ends an instance. The interference it suffers within time interval $[9, 15]$ from τ_1 and τ_2 is already integrated to $\bar{w}_3$, since the interference has been bounded with the activations number multiplied by the cost, and since the next τ_1 or τ_2 activation is not before time 15. The value of $\bar{w}_3$ is then increased by one period ($29 - 14 = 15$) since we now consider the next deadline, and decreased by the interference of τ_1 and τ_2 in the time interval $[15, 30]$. We have $S_3(9) = (2 + 15 - 6 - 5) - 2 = 3$.

Remarks on the interference computation. In this example, the interference computation of τ_1 and τ_2 on τ_3 is not hard because when the τ_3 deadline is reached all the begin instances are ended. The interference caused by a task is a multiple of its activation number.

If it was not the case, DASS should compute the exact interference, that is why the constant in the needed computation is high. With MASS, only an upper bound is used. This approximation permits to only consider the tasks activation at time instant grater than or equal to the deadline.

6.4. Discussion on the differences

With the DASS algorithm, it is some times possible to obtain a more accurate bound. However we see that the gap is between the two bounds (the one obtained with DASS and the one obtained with MASS) is fill in each time a periodic task ends an execution. At this expense, the bound is obtained with a much lesser greedy algorithm. Since the overhead of the algorithm has to

be included in the worst case execution time of each periodic task, it result in a lower schedulability bound for systems using DASS than for systems using MASS. It is so precisely when a more accurate bound should be necessary that DASS could lead to make the system non feasible while it could had been with MASS.

7. Framework to write a server

Task server policies only differ by the way they use and replenish their capacity. In every case, they can be implemented as an hard real time task which access to a soft real time task queue. Then this task takes the first soft task and runs it until the server capacity is consumed.

We identified three implementation issues: the server has to monitor the CPU consumption of served tasks in order to keep up to date its capacity, it has to be able to suspend the served task when the capacity fall to zero, and it has to be able to wake up when the next capacity replenishment instant occurs.

7.1. How does it work

We saw that an event can be implemented with RTSJ using the `AsyncEvent` class (AE). Its treatment can be implemented with the `AsyncEventHandler` class (AEH). Several events can been associated to a unique treatment, and several treatments can been associated to a unique event. This is a many to many relation. The schedulable object is the treatment, not the event itself.

Based on this model, we propose the class `ManageableAsyncEventHandler` (MAEH) to represent a treatment that can be handled by a server. This class does not implement the `Schedulable` interface, since the code its encapsulate is not destinate to be scheduled directly by the RTJVM, but within a server. Then we also propose an subclass to `AsyncEvent` we call `ManageableAsyncEvent` (MAE). This kind of event still can be associated to regular AEH, but can also be associated to MAEH.

The server itself is implemented with the new class `AbstractUserLandTaskServer` which implements `Schedulable`. This abstract class contains the methods for the queue gesture.

When the `fire()` method is called on an MAE, the AEH bounded to it are activated with their respective priorities, and its MAEH are enqueued on the queues of the server they are attached to.

The BS duplication can be done with the event triggering. The MAEH encapsulate four private objects:

- an `Interruptible` which is the code of the treatment;
- an `AsynchousInterruptedException` to interrupt the duplication if the event can be served within the server;
- a `AsyncEvent` in order to trigger the duplication;

- an `AsyncEventHandler` associated to the lowest system priority which use the asynchronous interrupted exception to call the run() method of the interruptible object.

We also add a boolean to permits the activation or the deactivation of this mechanism. When an MAEH is added to a server queue, the `fire()` method of the AE is called. This activate the AEH with the low priority. This AEH use the method `doInterruptible()` of the exception to execute the code of the `Interruptible`. If this method ends correctly, the MAEH is dequeued from the server. A public method permits to trigger the exception from the server if it is this last which success to treat the MAEH at first.

To resume, we propose the following classes to model a server, an event and its handlers:

ManageableAsyncEvent. This class inherits from AE. We add a list of MAEH objects and we overrun the `addHandler(AsyncEventHandler h)` method in order to permits the add of MAEH. Finally we overrun the `fire()` method to make it add the MAEH to the queue of the associated server.

ManageableAsyncEventHandler. This class contains an `Interuptible` field. It also has a reference on the server to which the handler is attached. When an MAE bounded to it is fired, its reference is added in the server queue, and if the duplication BS is activated, an AEH is activated with the system lowest priority.

AbstractUserLandTaskServer. This is an abstract implementation to provide the queue gesture methods.

7.2. How to write a PS

The PS is a periodic task. It starts each instances with its full capacity and can begin to serve the event enqueued. If the queue is empty, it looses its remaining capacity.

It so can be implemented with a class `UserLandPollingTaskServer`. Since there is no multiple inheritance with Java, this class cannot inherit from `RealtimeThread`. However, we can use a private field to store a reference on a `RealtimeThread`. All the non implemented method of the class `AbstractTaskServer` can be delegated to the instance referenced by this field.

7.3. How to write a DS

The DS can be integrated to the feasibility analysis, but is not properly a periodic task. Its capacity is restored periodically, but it can be activated at any time if it still has capacity.

Its implementation is quite similar to the PS one, except that the `Schedulable` object encapsulated could no more be a `RealtimeThread`, but should be an `Async_Event_Handler`. This handler is bounded to a special `AsyncEvent` fired

by a `PeriodicTimer` for its periodic capacity replenishment. This special event is also fired each time an aperiodic task is added in the server queue.

Another particularity of the DS is that it can have at a given time t a capacity greater that its periodically refreshed capacity. Indeed, if it remains x time unit of capacity, then $x$tu before the periodic replenishment, the server disposes of $x + C_s$ tu of capacity. If a task is enqueued with a cost greater than the server capacity, it should be necessary to wake up the server x tu before the replenishment of its capacity. Again, this can be done with a `Timer` which fired the special event.

8. Performances evaluation

8.1. Simulations

The validation of the effectiveness of our slack estimation is the first purpose of our simulations. Despite the simplicity of our algorithm induced by the wanted low overhead, we expect results comparable to an exact slack computation based algorithm. The targeted performance metric is the mean response time of the aperiodic requests.

To achieve this goal, we simulate the same systems with aperiodic tasks served according to our slack estimation and with the same aperiodic tasks (same release times and same costs) served according to the approximation of *DASS* and with an exact computation of the available slack.

The second expecting result is the validation of the userland exploitation of slack time. The other available algorithms that does not need to customize the scheduler are the Background Scheduling (BS) the modified PS (MPS) and the modified DS (MDS). So we compare the result obtained with these algorithm and the slack stealers approaches.

Finally, we also want to determine the best queue policy through our simulations.

8.1.1. Methodology

We measure the mean response time of soft tasks with different aperiodic and periodic loads.

First, we generate groups of periodic task sets with utilization levels of 30, 50, 70 and 90%. The results presented in this section are averages over a group of ten task sets. For the same periodic utilization, we repeat generations over a wide range of periodic task set composition, from systems composed by 2 periodic tasks up to systems composed by 100 periodic tasks.

The periods are randomly generated with an exponential distribution in the range [40-2560] time units. Then the costs are randomly generated with an uniform distribution in the range [1-period] In order to test systems with deadlines less than periods, we randomly generate deadlines with an exponential distribution in the range [cost-period]. Priorities are assigned assuming a deadline monotonic policy.

Non feasible systems are rejected, the utilization is computed and systems with an utilization level differing by less than 1% from that required are kept.

For the polling server, we have to find the best period T_s and capacity C_s couple. We try to maximize the system load U composed by the periodic load U_T and the server load U_S.

$$\begin{aligned} U &= U_T + U_S \\ U &= \sum \frac{C_i}{T_i} + \frac{C_s}{T_s} \end{aligned}$$

A feasible system load being bounded by 1, we have Equation 15.

$$\frac{C_s}{T_s} \leq 1 - \sum \frac{C_i}{T_i} \tag{15}$$

To find a lower bound C_s^{min} of C_s, we first set the period to 2560 (the maximal period). We then search the maximal value for C_s in $[1, 16]$ in order to keep a feasible system. This maximal value is our lower bound on C_s.

Then we seek the lower possible T_s value in $[C_s^{min}/(1 - \sum C_i/T_i), 2560]$. For each T_s value tested, we try decreasing values of C_s in $[C_s^{min}, T_s(1 - \sum C_i/T_i)]$.

Note that it is possible to find a capacity lower than the maximal aperiodic tasks cost. In such cases, since we have to schedule the aperiodic tasks in one shot, we have no solution but to background scheduling the tasks with a cost greater than the server capacity.

For the deferrable server, the methodology is similar, except that since the server has a bandwidth preservation behavior, we do not try to minimize the period and we can search the maximal C_s value in $[1, 2560]$.

Finally, we generate groups of ten aperiodic task sets with a range of utilization levels (plotted on the x-axis in the following graphs). Costs are randomly generated with an exponential distribution in the range [1-16] and arrival times are generated with an uniform distribution in the range [1-100000]. Our simulations end when all soft tasks have been served.

8.1.2. Real-time Simulator

We use a Java program which can simulate the execution of an event-based real-time system and display a temporal diagram of the simulated execution.

This tool is distributed under the General Public License GNU (GPL), and can be found on the following web page: `http://igm.univ-mlv.fr/~masson/RTSS`

8.1.3. Results

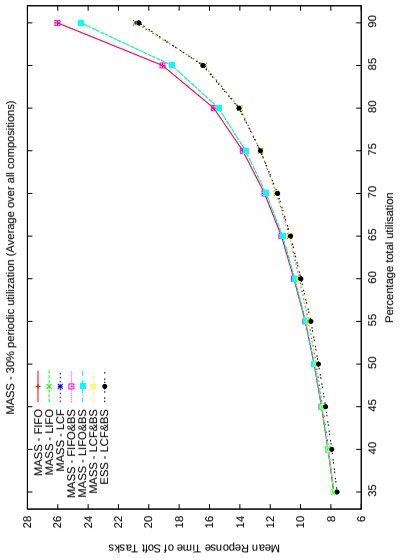


Figure 7: MASS, 30% load

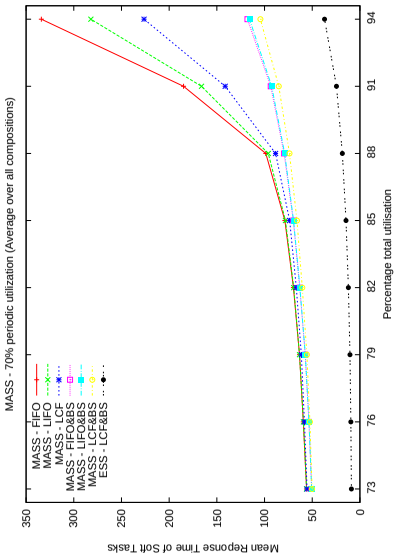


Figure 9: MASS, 70% load

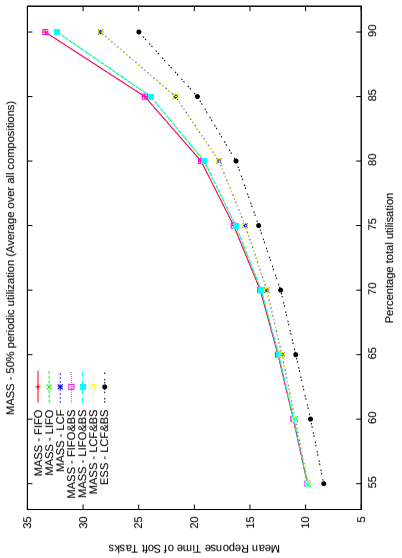


Figure 8: MASS, 50% load

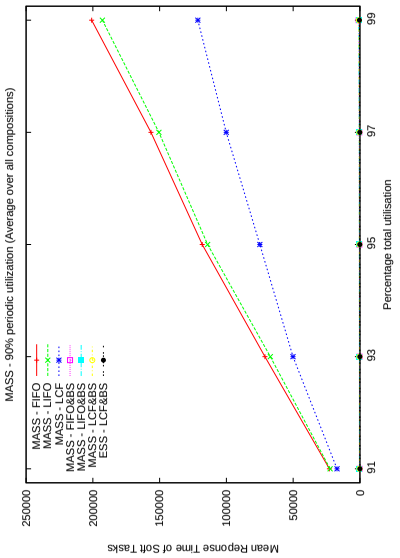


Figure 10: MASS, 90% load

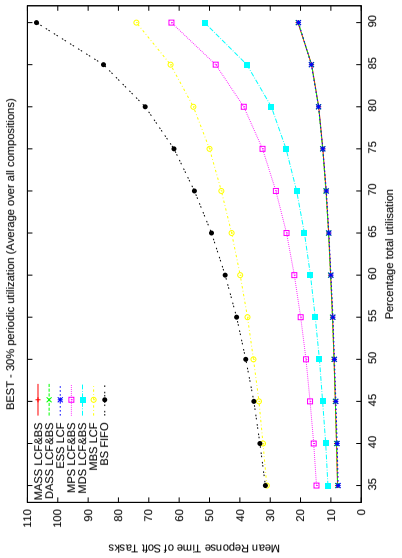


Figure 11: Best policies, 30% load

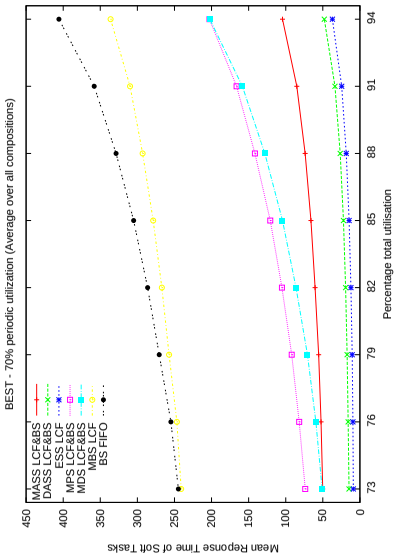


Figure 13: Best policies, 70% load

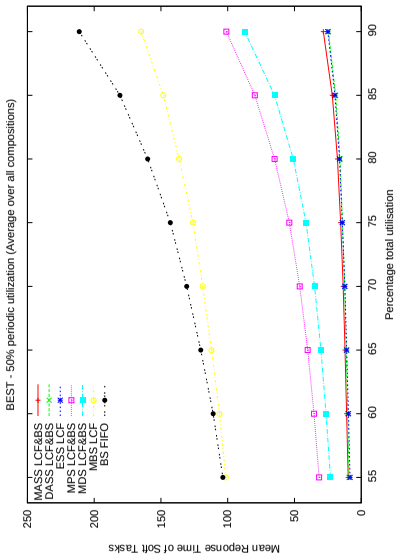


Figure 12: Best policies, 50% load

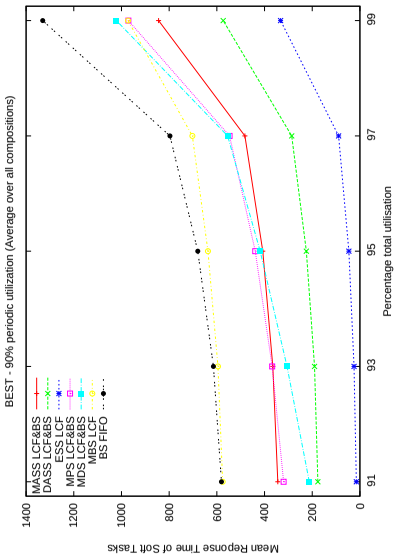


Figure 14: Best policies, 90% load

Figures 7 to 14 present our simulations results. On these figures, ESS, $DASS$ and $MASS$ refer to our slack stealer modified algorithm associated respectively with an exact computation of the available slack time, the slack time approximation given by $DASS$ and our approximation of the available slack time. MPS and MDS designate the modified polling server and deferrable server. Finally BS designates the background scheduling associated with a $FIFO$ queue policy and MBS a modified background server which cannot begins a task if a previously started task has not completed. The notation $X\&BS$ refers to the policy X with a BS duplication.

Figures 7 to 10 show the $MASS$ results for all periodic composition systems. We notice that our algorithm performs much better than BS for all policies if the periodic load is low (see Figure 7 and 8). However, when the periodic load increases, the BS duplication became unavoidable. With the extreme 90% load (Figure 10), the non BS-duplicated curves are not viewable on the same graph than the other and are completely out performed by the BS. This phenomena was expected and was the reason of the duplication introduction: when the load is too high, there is never enough slack to serve a request in one shot.

The second thing to note is that the queue policy which offers the best results is the LCF one. Due to space limitations and clarity purpose, we cannot put all our results in this paper, but this trend is confirmed by simulations on MPS, MDS, $DASS$ and ESS. This is for sure amplified by the one shot execution limitation. The shorter a task is, the greater is the probability to have quickly enough slack to schedule it completely.

Figures 11 to 14 present the results of the best queue policy for each algorithm (MPS, MDS, $MASS$, $DASS$ and ESS). For ESS, since the time complexity is dependent on the number of task, the results do not include systems composed of more than 40 periodic tasks. For all load conditions, servers bring real improvement comparing to BS. The MDS offers better performances than the MPS. Then, $MASS$ performs better than the servers, $DASS$ better than $MASS$ and ESS better than $DASS$. For systems with periodic loads of 30% and 50%, results obtained with $MASS$, $DASS$ and ESS are quite similar. Considering the differences between the time complexities of these algorithms (constant, linear and pseudo polynomial), this is a very satisfying result. However $MASS$ performances degrade when periodic load increases. Nevertheless $MASS$ remains the best userland-implementable algorithm even for systems with a periodic load of 90%.

Moreover, these simulations do not take into account the time overhead of the implementations. These systems where DASS performs much better than MASS are precisely the ones where the lower overhead of MASS could plays an important role: if DASS theoretically can schedule better these systems, it can in practice make them unscheduleable.

8.1.4. Conclusions

Our simulations showed that the BS duplication is essential.

Due to space limitation, we do not put the result here, but simulations also demonstrate that MPS and MDS are as good as regular PS and DS. In some

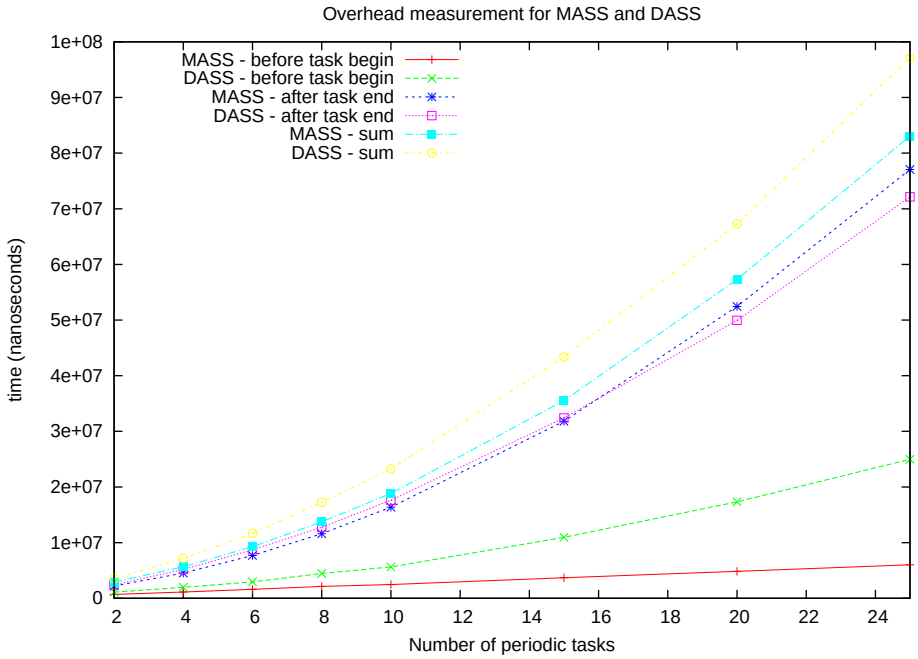


Figure 15: Comparative overheads of MASS and DASS

situations, we even obtain better results with modified policy than with regular ones. The explanation is that the modifications result in delaying some costly events, so shorter events which occur later can be immediately served, which result in shorter response times.

Until important load condition, MASS remains competitive against DASS. Performances degrades only when systems are very overloaded. However, the simulations does not take into account the time overhead in the system feasibility analysis. Due to a lower overhead, MASS gives a higher schedulability bound, which as to be balanced with the theoretically possible results.

8.2. Executions

8.2.1. On JamaicaVm

The DASS and MASS implantations that we propose only differ by the added code at the beginning and at the end of each instance of each periodic task.

We so have measured the time passed to execute this code with the two algorithms. To perform these measures, we used the jamaica virtual machine on a real-time linux free kernel 2.6.9 no an 1,22 GHz Intel(R) Pentium(R).

Since the time complexity of added code is constant or linear in the number of tasks, we have performed the measures with a number of tasks varying between 2 and 25.

Each task executes five instances. The period of each task is randomly generated in the interval $[5, 10]$ and its cost in the interval $[2, T_i]$. The first generated task has the lowest RTSJ priority (11), the second 12 and so on.

Figure 15 presents the obtained results. The cumulated time passed in the two methods is more important for DASS. The time passed in the added codes increase with the number of tasks and so the difference between the times obtained with the two algorithms.

Even for a low number of task (2), these costs are measurable and so have to be integrated in the feasibility analysis. We can see that until 10 tasks, the time passed in the added cost at the end of the instances is more important for DASS than for MASS. However this change when there is more tasks. This is due to the fact that the code added for DASS includes more instructions for each loop passage. Even if the two algorithms have the same complexity, MASS has to loop on all the task whereas DASS only consider tasks with higher priorities.

The total overheads in time for each instance is more important with DASS than with MASS. So the feasibility bound of a system will be lower with DASS than with MASS. So even if it does not perform at well at DASS, MASS can be used with more systems than DASS. This justify the use of MASS against DASS on systems with a high periodic load, even is theoretical simulation tends to show that DASS outperforms MASS. Indeed these kind of systems are precisely the ones where adding a two large overhead lead to affect the feasibility.

8.2.2. On lejosRT

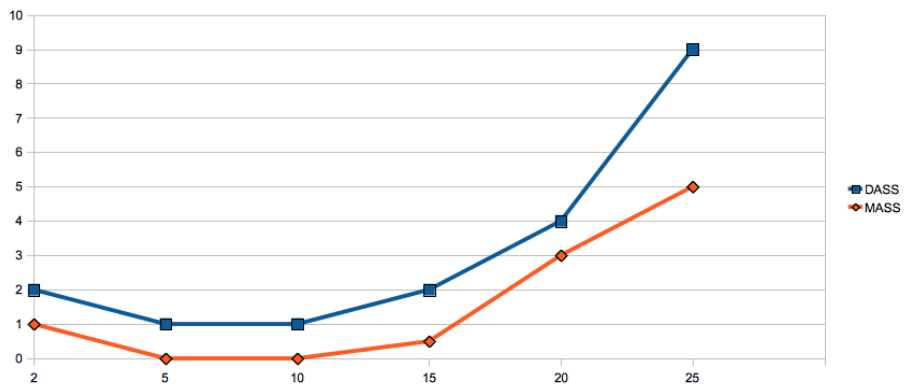


Figure 16: System overheads with DASS and MASS depending on task number

LejosRT is an open source project forked from Lejos, which is an alternative firmware for the Lego Mindstorm NXT brick. We implement DASS and MASS on LejosRT, and perform some overhead measure.

We consider a program with n real-time threads with the same release parameters (period and deadline) but with n different priorities. Moreover we shift start times in order to start first the thread with the lowest priority and last the one with the highest priority. That permits to maximize the number

of preemptions. The period of the thread with the lowest priority is set shorter than the other one in order to have this thread preempted twice be each other ones. We measure the needed time to execute two instances for each thread. The system load is then maximal (100%).

Figure 16 presents the results obtained for a range of thread number from 2 to 25. Each test was performed several times and the values reported in Y axis are the average differences between the time execution of the program on LejosRT without any slack computation and with the concerned algorithm (respectively DASS and MASS). On the X axis is reported the number of thread.

We can note that the MASS overhead is always lower than the DASS one. This is the expected result, that this experiment confirmed on real implementations. Moreover, the more thread we have, the higher the measured difference is.

9. Feed back and Suggestions to Improve the RTSJ

We present in this Section three proposition to improve the RTSJ. The first is the measurement of CPU time consumption, which is a primordial issue. The second is linked to our algorithms, but can be serve in a lot of domain: its an easiest and integrated way to automatically trigger code execution at the beginning and at the end of periodic tasks. The last one is a remark on the integration of the feasibility analysis within the RTSJ.

9.1. temps cpu

A method that returns the current time consumption of a task is missing, but is the point seven of the JSR 282 :

> *7. Add a method to* `Schedulable` *and* `ProcessingGroupParameters`*, that will return the elapsed CPU time for that schedulable object (if the implementation supports CPU time)*

This feature depends on the ability of the underlying operating system to provide this information. However, we have seen that we can obtain this information at user land level by adding operations at the begin and the end of each real-time tasks. Indeed, a stack can be maintained with the previous executing task on the top, and its cost refreshed at each context switches.

So this mechanism could be integrated in the `RealtimeThread` class, with a setter to enable/disable it.

9.2. Adding code before and after periodic instances

We have seen that this feature can be use to obtain the CPU time consumption even if the underlying operating system does not propose this feature, and to implement some advance mechanisms such as DASS or MASS.

The added execution time cost then can be take into account within the feasibility analysis process. This implies that class `Scheduler` has acces to this execution time.

Moreover, some treatment are only usable if all task actually participate. If there is in the system one task that do not cooperate to the user land CPU time consumption monitoring, for example, the value maintain by the mechanism is wrong. It also sometime necessary to make these code sections non preemptible to ensure the correctness of the mechanism. Finally, it could be useful to generalize the process to any kind of context switches.

For all these reason, it seems appropriate to delegate these code sections execution to the class `Scheduler`, with the addition of the following method:

- `void addContextSwitchHandler(`
  `AsyncEventHandler handler, boolean beforeSchedulable,`
  `boolean afterSchedulable, boolean afterEachContextSwitch);`

9.3. Feasibility Analysis

Les méthodes permettant de réaliser l'analyse de faisabilité sont toutes situées dans les classe `Scheduler` et `RealtimeThread`. En fait, les méthodes de `RealtimeThread` sont des raccourcis faisant appel à celles de `Scheduler`. Par conséquent, pour changer l'algorithme d'analyse de faisabilité utilisé, par exemple pour prendre en considération le cas de l'utilisation d'un serveur ajournable, il faut surcharger l'ordonnanceur.

All the methods concerning the feasibility analysis are in the class `Scheduler` and `RealtimeThread`. More precisely, the methods in `RealtimeThread` are convenient methods to call back the one in `Scheduler`. Consequently, in order to change the way the system is analyzed (eg to take into account the presence of a DS) one have to change the scheduler.

This is not justified. Indeed, one can want to analyze the system in a different way without changing its behavior. Fonction of the kind of targetted applications, a simple load test can be sufficient. One can also request the assurance of k deadline met on m instances ($(m,k)firm$ model [14]).

Of course, it remains possible to extend the scheduler class and to only overload the feasibility analysis methods. However, it is not satisfying in an object oriented design point of view.

We propose the creation of an interface `FeasibilityAnalysis`. Then the FA related methods in Scheduler can simply be delegated to an instance of `FeasibilityAnalysis`. Associated new methods in `Scheduler` can be added: `setFeasiblityAnalysis( FeasibilityAnalysis fa)` and `getFeasiblityAnalysis( FeasibilityAnalysis fa)`.

We can then propose at least two subinterfaces: `NecessaryTest` and `Sufficient-Test`.

10. Conclusions

We studied the jointly scheduling of hard periodic tasks with soft aperiodic events problem, where the response times of soft tasks have to be as low as possible while the warranty to meet their deadlines has to be given to hard tasks.

We presented the state of the art and discussed the implementability of proposed solutions under the real-time specification for Java (RTSJ), without changing the scheduler.

This led us to adapt existing algorithms to operate at a user land level in the system: the algorithm MPS and MDS and showed how to implement DASS.

We proposed some optimizations and counter measures in order to balance the lost of performances: the intelligent gesture of the waiting task queue and the BS duplication.

Finally we set up an approximate slack stealer algorithm specifically designed to take into account RTSJ restrictions: MASS.

We proposed new classes to extend the RTSJ API's to implement these mechanisms and some minor modification suggestions to existing ones as a feed back from our RTSJ experiences.

We demonstrated the efficiency of the modified algorithms through extensive simulations and their implementability on available RTSJ compliant virtual machine by an overhead measure in real situation with the RTSJ JamaïcaVM from Aïcas. We also measured the overhead on LejosRT, an RTSJ compliant firmware for Lego Mindstorms NXT in development.

If MASS was set up in order to address the mixed scheduling problem, slack time analysis is useful in another domains, like the scheduling using Dynamic Frequency/Voltage Scaling (DFS/DVS) abilities of modern CPUs. The use of MASS against DASS or another existing algorithms should be studied in future work.

References

[1] J. P. Lehoczky, L. Sha, J. K. Strosnider, Enhanced Aperiodic Responsiveness in Hard Real-Time Environments, in: Proceedings of the Real-Time Systems Symposium, IEEE Computer Society, San jose, California, ISBN 0-8186-0815-3, 110–123, 1987.

[2] B. Sprunt, J. P. Lehoczky, L. Sha, Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm, in: Proceedings of the Real-Time Systems Symposium, Huntsville, AL, USA, 251–258, 1988.

[3] B. Sprunt, L. Sha, J. P. Lehoczky, Aperiodic Task Scheduling for Hard Real-Time Systems, Real-Time Systems: The International Journal of Time-Critical Computing Systems 1 (1989) 27–60.

[4] J. P. Lehoczky, S. Ramos-Thuel, An optimal algorithm for scheduling soft-aperiodic tasks fixed priority preemptive systems, in: proceedings of the 13th IEEE Real-Time Systems Symposium, Phoenix, Arizona, 110–123, 1992.

[5] T.-S. Tia, J. W.-S. Liu, M. Shankar, Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems, Real-Time

Systems: The International Journal of Time-Critical Computing Systems 10 (1) (1996) 23–43, ISSN 0922-6443, doi:\bibinfo{doi}{http://dx.doi.org/10.1007/BF00357882}.

[6] R. I. Davis, On Exploiting Spare Capacity in Hard Real-Time Systems, Ph.D. thesis, University of York, 1995.

[7] H. Chetto, M. Chetto, Some results of the Earliest Deadline Scheduling algorithm, IEEE Transactions on Software Engineering 15 (10).

[8] M. Silly-Chetto, The EDL Server for Scheduling Periodic and Soft Aperiodic Tasks with Resource Constraints, The Journal of Real-Time Systems 17 Kluwer Academic Publishers.

[9] S. Ramos-Thuel, J. P. Lehoczky, On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems, in: Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS '93), 1993.

[10] R. I. Davis, K. Tindell, A. Burns, Scheduling Slack Time in Fixed Priority Pre-emptive Systems, in: Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS '93), 222–231, 1993.

[11] A. Burns, A. Wellings, Processing Group Parameters in the Real-Time Specification for Java, in: On the Move to Meaningfull Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems, vol. LNCS 2889, Springer, 360–370, URL `http://www.cs.york.ac.uk/rts/cgi-bin/bibtex/bibtex.pl?key=R:Burns:2003g`, 2003.

[12] A. Wellings, M. Kim, Processing Group Parameters in the Real-Time Specification for Java, in: proceedings of JTRES 2008, 2008.

[13] B. Sprunt, Aperiodic Task Scheduling for Real-Time Systems, Ph.D. thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 1990.

[14] P. Ramanathan, M. Hamdaoui, A Dynamic Priority Assignment Technique for Streams with (m, k)-Firm Deadlines, IEEE Trans. Comput. 44 (12) (1995) 1443–1451, ISSN 0018-9340, doi:\bibinfo{doi}{http://dx.doi.org/10.1109/12.477249}.